Using the Fast Fourier Transform Library FFTW

Danko Antolovic
Pervasive Technology Labs, Indiana University
11/2005

Overview of FFTW

FFTW is a library of C functions which compute discrete Fourier transforms. The library is fairly comprehensive: it computes complex and real Fourier transforms in any number of dimensions, and it has single-precision and double-precision forms. FFTW was first developed for Unix, but a Windows implementation is available as well. It is distributed as free software, under GNU General Public License.

FFTW utilizes the so-called Fast Fourier algorithm, which can improve the speed of the computation, dependent upon the number of input data points. This document demonstrates how to use FFTW in your program, and illustrates the concepts behind FFTW, by guiding you through a typical example. A comprehensive manual and installation instructions can be found at the FFTW web site:

http://www.fftw.org/

Theoretical background

In order to use this library effectively, you should have an understanding of the basic concepts of the Fourier transformation. This technique has applications in virtually every area of science, from quantum field theory to computer graphics. The transformation comes in several varieties, and textbook accounts sometimes differ in minor convention details, but the idea behind it is remarkably simple: a variable quantity, say a function of time, $f(t)$, can be approximated by a sum of harmonic functions (sines and cosines) of increasing frequencies. The higher the frequency, the better (more detailed) the approximation.

Not all frequencies contribute equally. For example, if your $f(t)$ is a 100 MHz clock signal, sine/cosine waves of just that frequency will make the strongest contribution, of course. But higher frequency waves will contribute also, and the crucial point is this: if you know the contributions of all frequencies, that knowledge is equivalent to knowing $f(t)$; nothing is lost. We usually write these contributions collectively, as a function of frequency $F(\omega)$, and call this function the Fourier transform of $f(t)$.

The transformation has an inverse, which is very similar, except that it transforms $F(\omega)$ back into $f(t)$. Both transformations are linear, so that, if $a$ and $b$ are any constants, the transformations take $a f(t) + b g(t)$ into $a F(\omega) + b G(\omega)$ and back again.

Intuitively, it is best to think of the Fourier transformation as approximating a function by sines and cosines. However, the actual formulas become much simpler and more general if the transformation is re-cast as an approximation by a sum of complex exponentials, which we can do thanks to Leonhard Euler's remarkable result:

$$\exp(ix) = \cos(x) + i\sin(x)$$

In this new variety, which is called the complex Fourier transformation, the transformed function "f" can be complex, and its transformation "F" is generally a complex function as well. All the basic properties (the inverse, linearity, convolution etc.) remain the same.

One property of the Fourier transformation that is very useful in graphics and imaging is known as the convolution theorem. Technically, a convolution of two functions is an integral of their product, where one function is displaced relative to the other. For example, a Gaussian or Gabor filter is convoluted with an image in the neighborhood of a pixel, i.e. at some displacement from the origin. In order to filter the entire image, you must slide the filter around and convolve it with the image around every pixel.
The convolution theorem states that the Fourier transformation of the convolution integral is the product of the transforms of the two functions. That means that we can transform the image, and the filter (disregarding the displacement), multiply the two, and inverse-transform the product. The result is the filtered image, all at once, without the need to multiply and sum around every pixel. The price that we pay for this simplification is in transforming the functions back and forth, but with a fast Fourier algorithm this technique can save a great deal of computational time.

There are many good descriptions of the Fourier transformation in the literature. Here are some references, to help you become familiar with the topic:

General description:
http://mathworld.wolfram.com/FourierTransform.html

Discrete Fourier Transform (the variety which transforms discrete arrays, instead of continuous functions):

Fast Fourier Transform (a highly successful numerical technique for calculating the transforms):

Convolution theorem:
http://www-structmed.cimr.cam.ac.uk/Course/Convolution/convolution.html

An overall reference on computer graphics and imaging, with good accounts of the Fourier techniques:

Implementation Example

Once you have FFTW installed on your system, link to the library file (fftw3, or whatever variation applies to your particular installation), and include the usual header file in your code:

```
#include <fftw3.h>
```

The FFTW data types and functions are prototyped in this header. All names of data types and functions are prefixed with "fftw_" (double precision) or "fftwf_" (single precision, float), and the function names indicate which variant and dimension of the transform is being computed. In this example, we use the single-precision, discrete transform in two dimensions.

First, we define pointers to (fftwf_) complex arrays, which will serve as input and output to the Fourier transforms. We also define the pointer to a data structure called the "plan", which essentially contains information that pertains to the dimensions of input and output arrays, and which is reused until these dimensions change.

```
fftwf_complex *in, *out;
fftwf_plan g;
```

We use the fftwf-specific version of malloc, to allocate memory for the input and output arrays:

```
in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * ROWS*COLS);
out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * ROWS*COLS);
```

Next, we invoke the function which creates the Fourier transform plan. From the name of the function, we see that this is a floating-point (single precision), discrete transform (not the continuous kind), in two dimensions. By default, the
transformation is complex. The function’s arguments specify the sizes of the 2-D array, and the I/O arrays (declared and allocated above); also, this is the forward transform (not the inverse), and the level of the plan's optimization is set high. See the FFTW documentation for a more detailed description of the plans.

```c
    g = fftwf_plan_dft_2d(ROWS, COLS, in, out, FFTW_FORWARD, FFTW_MEASURE);
```

Suppose now that we want to repeatedly calculate Fourier transforms of Gabor masks of the same size, but of different orientation. Within the loop that runs over orientations, generate the sine and cosine Gabor masks and pad them with zeros to the size of your image. Suppose that the cosine mask is in the array `C[ROWS][COLS].`

Now, the `fftwf_complex` array "in" is actually of the dimension `in[ROWS*COLS][2]`, where the first index runs over the image, and the second distinguishes the real and imaginary components (remember that the Fourier transform lives most comfortably in the complex plane!). We need to transfer our Gabor mask (which is real) into the real part of "in", and zero out the imaginary part, something like this:

```c
    // copy the matrix C to the real part of FFTW input
    k = 0;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            in[k][0] = C[i][j];
            in[k++][1] = 0;
        }
    }
```

Only now do we actually calculate the transform, by executing its plan. Notice how generic this call is: all the specifics of the transform were set up in the plan itself, and in the input array.

```c
    fftwf_execute(g);
```

The remaining step is to extract the result. Transforms of functions of two variables (or 2-D arrays) are themselves two-dimensional, so we must unscramble the 1-D output array "out."

```c
    // copy the real part of FFTW output back to C
    k = 0;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            C[i][j] = out[k++][0];
        }
    }
```

In this step, we took the real part of the transform and placed it back in C. What about the imaginary part?

Complex Fourier transform treats the cosine waves as real, and sine waves as imaginary. Remember that sines and cosines are out of phase by 90 degrees; this can be viewed as a 90 degree rotation from real numbers to the imaginary ones. A wave of any phase (a combination of sine and cosine) can be represented by a complex number (a combination of real and imaginary part), and will have a complex Fourier transform.

However, we started with a cosine mask, whose wave has only a cosine component, that is, a phase of zero represented by real numbers. The wave is multiplied by a Gaussian curve, but that is an even function and does not change the phase of the wave. We conclude that the entire sine part of the transform must vanish, and we simply ignore it. In fact, this is a good diagnostic test: if the imaginary part of a cosine wave's transform is NOT zero (or very, very small) something is wrong with the implementation!

The transformation of a sine Gabor mask follows analogous steps. Since the plan does not care about the actual input function, we can use the same plan "g", and the same I/O arrays. The sine mask is real, so we place it in the real part of the FFTW input, and zero out the imaginary part. We execute the plan, and the result should reside in the imaginary part of the output array.
Another way of looking at the above reasoning is to remember that cosines are even: \( \cos(x) = \cos(-x) \), while sines are odd: \( \sin(x) = -\sin(-x) \). A purely even or odd function must be represented by functions of the same parity, and that is why cosine waves can't have sine components.

Calculating the inverse Fourier transform follows the same steps as the forward transform. Set up the plan for the inverse transform:

```c
fftwf_plan inv_g = fftwf_plan_dft_2d(ROWS, COLS, inv_in, inv_out,
  FFTW_BACKWARD, FFTW_MEASURE);
```

and notice the option `FFTW_BACKWARD` in the plan. The formulas for the forward and inverse transforms differ only in the sign of the exponent.

Finally, the FFTW library has an idiosyncrasy that you should be aware of: it omits the usual normalization constants from the calculation. These constants are \( 1/\sqrt{ROWS} \) in the 1-D case, and \( 1/\sqrt{ROWS \times COLS} \) for two dimensions, and the FFTW's results must be multiplied with them at the end. If you transform an array back and forth, the result should be identical to the original array, since these two operations are, in fact, inverses of each other. In the FFTW case, a 2-D array will end up being multiplied by the factor \( (ROWS \times COLS) \).

At the end of the calculation, we free up the memory that was taken up by the Fourier transform:

```c
fftwf_destroy_plan(g);
fftwf_free(in);
fftwf_free(out);

fftwf_destroy_plan(inv_g);
fftwf_free(inv_in);
fftwf_free(inv_out);
```